Towards Automated Deduction in Blackmail Case Analysis with Forensic Lucid

Serguei A. Mokhov Joey Paquet Mourad Debbabi
Faculty of Engineering and Computer Science
Concordia University, Montréal, Québec, Canada,
{mokhov,paquet,debbabi}@encs.concordia.ca

Abstract

This work-in-progress focuses on the refinement of application of the intensional logic to cyberforensic analysis and its benefits are compared with the finite-state automata approach. This work extends the use of the scientific intensional programming paradigm onto modeling and implementation of a cyberforensics investigation process with the backtrace of event reconstruction, modeling the evidence as multidimensional hierarchical contexts, and proving or disproving the claims with it in the intensional manner of evaluation. This is a practical, context-aware improvement over the finite state automata (FSA) approach we have seen in the related works. As a base implementation language model we use in this approach is a new dialect of the Lucid programming language, that we call Forensic Lucid and in this paper we focus on defining hierarchical contexts based on the intensional logic for the evaluation of cyberforensic expressions.

Keywords: intensional logic, intensional programming, cyberforensics, Forensic Lucid, Lucid, finite-state automata

1 Introduction

Problem Statement. The first formal approach for event reconstruction cyberforensic analysis appeared in two papers [1, 2] by Gladyshev et al. that relies on the finite-state automata (FSA) and their transformation and operation to model evidence, witnesses, stories told by witnesses, and their possible evaluation. One of the examples the papers present is the use-case for the proposed technique – Blackmail Investigation. We aim at the same case to model and implement it using the new approach, which promises to be more friendly and usable in the actual investigator's work and serve as a basis to further development in the area.

Proposed Solution. We intend to show the intensional approach with a Lucid-based dialect to the problem is an asset in the field of cyberforensics as it is promising to be more practical and usable than the FSA. Since Lucid was originally designed and used to prove correctness of programming languages [3, 4], and is based on the temporal logic, functional and data-flow languages its implementation to backtracking in proving or disproving the evidential statements and claims in the investigation process as a evaluation of an expression that either evaluates to *true* or *false* given all the facts in the formally specified context. We will also attempt to retain the generality of the approach vs. building a problem-specific FSA in the FSA approach that can suffer a state explosion problem.

From the logic perspective, it was shown one can model computations (the basic unit in the finite state machines in [1, 2]) as logic [5]. When armed with contexts as first-class values and a demand-driven model adopted in the implementation of the Lucid-family of languages [6, 7, 8, 9, 10] that limits the scope of evaluation in a given set of dimensions, we come to the intensional logic and the corresponding programming artifact. In the essence, we model our forensic computation unit in the intensional logic and propose the ways to implement it in practice within an intensional programming platform [11, 6, 12, 8].

We see a lot of potential for this work to be successful and beneficial for cyberforensics as well as intensional programming communities.

Based on the parameters and terms defined in the papers [1, 2], we have various pieces of evidence and witnesses telling their own stories of the incident. The goal is to put them together to make the description of the incident as precise as possible. To show that a certain claim may be true, an investigator has to show that there are some explanations of evidence that agrees with the claim. To disprove the claim, the investigator has to show there is no explanation of evidence that agree with the claim [1]. The authors of the FSA approach did a proof-of-concept implementation of the algorithms in CMU Common LISP [1] that we target to improve by re-writing in a Lucid dialect, that we call Forensic Lucid. In this work we focus on the specification of hierarchical context expressions and the operators on them when modeling the examples. LIPS, unlike Lucid, entirely lacks contexts build into its logic, syntax, and semantics, thereby making the implementation of the cases more clumsy and inefficient (i.e. highly sequential). Our system [11] (not discussed here) offers distributed demand-driven evaluation of Lucid programs in a more efficient way and is more general than LISP's compiler and run-time environment.

Lucid Overview. Lucid [3, 4, 13, 14, 15] is a dataflow intensional and functional programming language. In fact, it is a family of languages that are built upon intensional logic (which in turn can be understood as a multidimensional generalization of temporal logic) involving context and demand-driven parallel computation model. A program written in some Lucid dialect is an expression that may have subexpressions that need to be evaluated at certain *context*. Given the set of dimensions $D = \{dim_i\}$ in which an expression varies, and a corresponding set of indexes or *tags* defined as placeholders over each dimension, the context is represented as a set of $<dim_i : tag_i>$ mappings and each variable in Lucid, called often a *stream*, is evaluated in that defined context that may also evolve using context operators [7, 9, 10, 16]. The generic version of Lucid, the General Intensional Programming Language (GIPL) [17], defines two basic operators @ and # to navigate (switch and query) in the contexts $\mathscr{P}$. The GIPL is the first[1] generic programming language of all intensional languages, defined by the means of only two intensional operators `@` and `#`. It has been proven that other intensional programming languages of the Lucid family can be translated into the GIPL [17].

1.1 General Intensional Programming System (GIPSY).

The GIPSY [18, 11, 19, 20, 12, 21, 22, 6, 8] is a platform implemented primarily in Java to investigate properties of the Lucid family of languages and beyond. It executes Lucid programs following a demand-driven distributed generator-worker architecture, and is designed as a modular collection of frameworks where components related to the development (RIPE[2]), compilation (GIPC[3]), and execution (GEE[4]) of Lucid programs are separated allowing easy extension, addition, and replacement of the components. This is a proposed testing and investigation platform for our *Forensic Lucid* language.

2 Forensic Lucid Overview

This section summarizes concepts and considerations in the design of the Forensic Lucid language, which is being studied through another use-case than related works [23, 24]. The end goal is to define our Forensic Lucid language where its constructs concisely express cyberforensic evidence as context

[1] The second being Lucx [9, 10, 16]

[2] Run-time Integrated Programming Environment, implemented in `gipsy.RIPE`

[3] General Intensional Programming Compiler, implemented in `gipsy.GIPC`

[4] General Eduction Engine, implemented in `gipsy.GEE`

of evaluations, which can be initial state of the case towards what we have actually observed as a final state in the FSM. The implementing system [6, 11, 12, 8] backtraces intermediate results to provide the corresponding event reconstruction path if it exists (which we do not discuss in this work). The result of the expression in its basic form is either *true* or *false*, i.e. "guilty" or "not guilty" given the evidential evaluation context per explanation with the backtrace. There can be multiple backtraces, that correspond to the explanation of the evidence (or lack thereof).

Properties. We define Forensic Lucid to model the evidential statements and other expressions representing the evidence and observations as context. An execution trace of a running Forensic Lucid program is designed to expose the possibility of the proposed claim with the events that lead to the conclusion. Forensic Lucid aggregates the features of multiple Lucid dialects mentioned earlier needed for these tasks along with its own extensions. The addition of the context calculus from Lucx (stands for "Lucid enriched with context" that promotes contexts as first-class values) for operators on simple contexts and context sets (`union`, `intersection`, etc.) are used to manipulate complex hierarchical context spaces in Forensic Lucid. Additionally, Forensic Lucid inherits many of the properties of Lucx, Objective Lucid, JOOIP (Java-embedded Object-Oriented Intensional Programming language), and their comprising dialects, where the former is for the context calculus, and the latter for the arrays and structural representation of data for modeling the case data structures such as events, observations, and groupings of the related data, and so on. (We eliminate the OO-related aspects from this work as well as some others to conserve space and instead focus on the context hierarchies, syntax, and semantics.) Hierarchical contexts are also following the example of MARFL [25] using a dot operator and by overloading @ and # to accept different types as their left and right arguments. One of the basic requirements is that the final target definition of the syntax, and the operational semantics of Forensic Lucid should be compatible with the basic Lucx and GIPL. This is necessary for compiler and and the run time system within the implementing system, called General Intensional Programming System (GIPSY) [6, 11, 8]. The translation rules or equivalent are to be provided when implementing the language compiler within GIPSY, and such that the run-time environment (General Eduction Engine, or GEE) can execute it with minimal changes to GEE's implementation.

Context. We need to provide an ability to encode the stories told by the evidence and witnesses. This will constitute the context of evaluation. The return value of the evaluation would be a collection of backtraces, which contain the "paths of truth". If a given trace contains all truths values, it's an explanation of a story. If there is no such a path, i.e. the trace, there is no enough supporting evidence of the entire claim to be true.

The context for this task for simplicity of the prototype language can be expressed as integers or strings, to which we attribute some meaning or description. The contexts are finite and can be navigated through in both directions of the index, potentially allowing negative tags in our tag sets of dimensions. Concurrently, our contexts can be a finite set of symbolic labels and their values that can internally be enumerated. The symbolic approach is naturally more appropriate for humans and we have a machinery to so in Lucx's implementation in GIPSY [26, 10]. We define streams of observations as our context, that can be a simple context or a context set. In fact, in Forensic Lucid we are defining higher-level dimensions and lower-level dimensions. The highest-level one is the *evidential statement*, which is a finite unordered set of observation sequences. The *observation sequence* is a finite *ordered* set of observations. The *observation* is an "eyewitness" of a particular property along with the duration of the observation. As in the FSA [2, 1], the observations are tuples of (P, min, opt) in their generic form. The observations in this form, specifically, the property P can be exploded further into Lucx's context set and further into an atomic simple context [16, 7]. Context switching between different observations is

done naturally with the Lucid @ context switching operator. Consider some conceptual expression of a storyboard in Listing 1 where anything in `[ ... ]` represents a story, i.e. the context of evaluation. `foo` can be evaluated at multiple contexts (stories), producing a collection of final results (e.g. *true* or *false*) for each story as well as a collection of traces.

```
foo @
{
  [ final observed event, possible initial observed event ],
  [                       ],
  [                       ]
}
```

Listing 1: Intensional Storyboard Expression

While the `[...]` notation here may be confusing with respect to the notation of `[dimension:tag]` in Lucid and more specifically in Lucx [16, 10, 7], it is in fact a simple syntactical extension to allow higher-level groups of contexts where this syntactical sugar is later translated to the baseline context constructs. The tentative notation of `{[...],...,[...]}` implies a notion similar to the notion of the "context set" in [16, 7, 10] except with the syntactical sugar mentioned earlier where we allow syntactical grouping of properties, observations, observation sequences, and evidential statements as our context sets. The generic observation sequence [1] can be expanded into the context stream using the *min* and *opt* values, where they will translate into index values. Thus, $obs = (A,3,0)(B,2,0)$ expands the property labels A and B into a finite stream of five indexed elements: *AAABB*. Thus, a Forensic Lucid fragment in Listing 2 would return the third A of the *AAABB* context stream in the observation portion of o. Therefore, possible evaluations to check for the properties can be as shown in Figure 1.

```
// Give me observed property at index 2 in the observation sequence obs
o @.obs 2
where
  // Higher-level dimension in the form of (P,min,opt)
  observation o;
  // Equivalent to writing = { A, A, A, B, B };
  observation sequence obs = (A,3,0)(B,2,0);
  where
    // Properties A and B are arrays of computations
    // or any Expressions
    A = [c1,c2,c3,c4];
    B = E;
    ...
  end;
end;
```

Listing 2: Observation Sequence With Duration

The property values of A and B can be anything that context calculus allows. The `observation sequence` is a finite ordered context tag set [10] that allows an integral "duration" of a given tag property. This may seem like we allow duplicate tag values that are unsound in the classical Lucid semantics; however, we find our way around little further in the text with the implicit tag index. The semantics of the arrays of computations is not a part of either GIPL or Lucx; however, the arrays are provided by JLucid and Objective Lucid. We need the notion of the arrays to evaluate multiple computations at the same context. Having an array of computations is conceptually equivalent of running an a Lucid program under the same context for each array element in a separate instance of the evaluation engine and then the

results of those expressions are gathered in one ordered storage within the originating program. Arrays in Forensic Lucid are needed to represent a set of results, or *explanations* of evidential statements, as well as denote some properties of observations. We will explore the notion of arrays in Forensic Lucid much greater detail in the near future work. In the FSA approach computations c_i correspond to the state q and event i that enable transition. For Forensic Lucid, we can have c_i as theoretically any Lucid expression E.

```
Observed property (context): A A A B B
        Sub-dimension index: 0 1 2 3 4

o @.obs 0 = A
o @.obs 1 = A
o @.obs 2 = A
o @.obs 3 = B
o @.obs 4 = B

To get the duration/index position:

o @.obs A = 0 1 2
o @.obs B = 3 4
```

Figure 1: Handling Duration of an Observed Property in the Context

In Figure 1 we are illustrating a possibility to query for the sub-dimension indices by raw property where it persists that produces a finite stream valid indices that can be used in subsequent expressions, or, alternatively by supplying the index we can get the corresponding raw property at that index. The latter feature is still under investigation of whether it is safe to expose it to Forensic Lucid programmers or make it implicit at all times at the implementation level. This is needed to remedy the problem of "duplicate tags": as previously mentioned, observations form the context and allow durations. This means multiple duplicate dimension tags with implied subdimension indexes should be allowed as the semantics of a traditional Lucid approaches do not allow duplicate dimension tags. It should be noted however, that the combination of the tag and its index in the stream is still unique and can be folded into the traditional Lucid semantics.

Transition Function. A transition function (described at length [1, 2] and the derived works) determines how the context of evaluation changes during computation. A general issue exists that we have to address is that the transition function ψ is usually problem-specific. In the FSA approach, the transition function is the labeled graph itself. In the first prototype, we follow the graph to model our Forensic Lucid equivalent. In general, Lucid has already basic operators to navigate and switch from one context to another, which represent the basic transition functions in themselves (the intensional operators such as @, #, `iseod`, `first`, `next`, `fby`, `wvr`, `upon`, and `asa` as well as their inverse operators[5]). However, a specific problem being modeled requires more specific transition function than just plain intensional operators. In this case the transition function is a Forensic Lucid function where the matching state transition modeled through a sequence of intensional operators. A question arises a of how to explicitly model the transition function ψ and its backtrace Ψ^{-1} in the new language. A possible approach is to use predefined macros in Lucid syntax [27]. In fact, the forensic operators are just pre-defined functions that rely on the traditional and inverse Lucid operators as well as context switching operators that achieve

[5]Defined further

something similar to the transitions. At the implementation level, it is the GEE that actually does the execution of ψ within GIPSY. In fact, the intensional operators of Lucid represent the basic building blocks for ψ and Ψ^{-1}.

Operational Semantics. As previously mentioned, the operational semantics of Forensic Lucid for the large part is viewed as a composition of the semantic rules of GIPL, Objective Lucid, and Lucx along with the new operators and definitions. The explanation of the rules and the notation are given in great detail in the cited works and are trimmed in this extended abstract due to shortage of space. The Objective Lucid semantic rules were affected and refined by some of the semantic rules of JOOIP [28]. We also omit the Objective Lucid and JOOIP semantic rules due to space limitation and defer them to another publication. The new rules of the operational semantics of Forensic Lucid cover the newly defined operators primarily, including the reverse and logical stream operators as well as forensic-specific operators. Refining the semantics of context set operators of Lucx, such as `box` and `range` are also a part of this work. We use the same notation as the referenced languages to maintain consistency in defining our rules.

3 Initial Blackmail Case Modeling

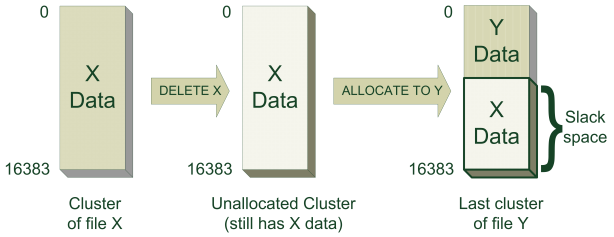

Figure 2: Cluster Data with the Blackmail Fragments

The case description in this section is from [2]. A managing director of some company, Mr. C, was blackmailed. He contacted the police and handed them evidence in the form of a floppy disk that contained a letter with a number of allegations, threats, and demands. The message was known to have come from his friend Mr. A. The police officers went to interview Mr. A and found that he was on holiday abroad. They seized the computer of Mr. A and interviewed him as soon as he returned into the country. Mr. A admitted that he wrote the letter, but denied making threats and demands. He explained that, while he was on holiday, Mr. C had access to his computer. Thus, it was possible that Mr. C added the threats and demands into the letter himself to discredit Mr. A. One of the blackmail fragments was found in the slack space of another letter unconnected with the incident. When the police interviewed the person to whom that letter was addressed, he confirmed that he had received the letter on the day that Mr. A had gone abroad on holiday. It was concluded that Mr. A must have added the threats and demands into the letter before going on holiday, and that Mr. C could not have been involved. In Figure 2 is the initial view of the incident as a diagram illustrating cluster data of the blackmail and unconnected letters.

Modeling the Investigation. In the blackmail example, the functionality of the last cluster of a file was used to determine the sequence of events and, hence, to disprove Mr. A's alibi. Thus, the scope of the model can be restricted to the functionality of the last cluster in the unrelated file. The last cluster model can store data objects of only three possible lengths: $LENGTH = \{0,1,2\}$. Zero length means that the cluster is unallocated. The length of 1 means that the cluster contains the object of the size of

Simplified model of the cluster:

possible data lengths: 0 1 2

LEFT_PART	*RIGHT_PART*

possible data values:
u – unrelated
t_1 – threats-obscured part t_2 – threats in slack
o_1 – other data left part o_2 – other data right part

Observed final state:
$LENGTH = 1$

(u) unrelated	(t_2) threats in slack

$LEFT\_PART = \{u, t_1, o_1\}$
$RIGHT\_PART = \{t_2, o_2\}$
$Q = LENGTH \times LEFT\_PART \times RIGHT\_PART$

Figure 3: Simplified Cluster Model

the unrelated letter tip. The length of 2 means that the cluster contains the object of the size of the data block with the threats. In Figure 3 is, therefore, the simplified model of the investigation.

Events. The state of the last cluster can be changed by three types of events:

1. Ordinary writes into the cluster:
   $WRITE = \{(u), (t_1), (o_1), (u, t_2), (u, o_2), (t_1, t_2), (t_1, o_2), (o_1, t_2), (o_1, o_2)\}$
2. Direct writes into the file to which the cluster is allocated (bypassing the OS):
   $DIRECT\_WRITE = \{d(u, t_2), d(u, o_2), d(o_1), d(t_1, t_2), d(t_1, o_2), d(o_1, t_2), d(o_1, o_2)\}$
3. Deletion of the file which sets the length of the file to zero:
   $I = WRITE \bigcup DIRECT\_WRITE \bigcup \{del\}$

Formalization of the Evidence. The final state observed by the investigators is $(1, u, t_2)$. Let O_{final} denote the observation of this state. The entire sequence of observations is then $os_{final} = (\$, O_{final})$. The observation sequence $os_{unrelated}$ says that the unrelated letter was created at some time in the past, and that it was received by the person to whom it was addressed is $os_{unrelated} = (\$, O_{unrelated}, \$, (C_T, 0, 0), \$)$ where $O_{unrelated}$ denotes the observation that the "unrelated" letter tip (u) is being written into the cluster. The evidential statement is then: $es_{blackmail} = (os_{final}, os_{unrelated})$.

Finding an Explanation of Mr. A's Theory. Mr. A's theory, encoded using the proposed notation, is $os_{Mr.A} = (\$, O_{unrelated-clean}, \$, O_{blackmail}, \$)$, where $O_{unrelated-clean}$ denotes the observation that the "unrelated" letter (u) is being written into the cluster and, at the same time, the cluster does not contain the blackmail fragment; $O_{blackmail}$ denotes the observation that the right part of the model now contains the blackmail fragment (t_2).

Explanations. There are two most logically possible explanations that can be represented by state machine. See the corresponding state diagram for the blackmail case in Figure 4.

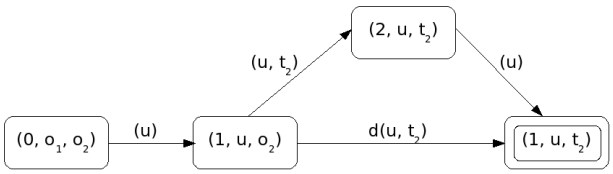

Figure 4: Blackmail Case State Machine

1. The first explanation:
   $\ldots \xrightarrow{(u)} (1,u,o_2) \xrightarrow{(u,t_2)} (2,u,t_2) \xrightarrow{(u)} (1,u,t_2)$
   - Finding the unrelated letter, which was written by Mr. A earlier;
   - Adding threats into the last cluster of that letter by editing it "in-place" with a suitable text editor (such as ViM [29]);
   - Restoring the unrelated letter to its original content by editing it "in-place" again.

     "To understand this sequence of events, observe that certain text editors (e.g. ViM [29]) can be configured to edit text "in-place". In this mode of operation, the modified file is written back into the same disk blocks that were allocated to the original file. As a result, the user can forge the file's slack space by (1) appending the desired slack space content to the end of the file, (2) saving it, (3) reverting the file back to the original content, (4) saving it again." [2]

2. The second explanation:
   $\ldots \xrightarrow{(u)} (1,u,o_2) \xrightarrow{d(u,t_2)} (1,u,t_2)$
   - The threats are added into the slack space of the unrelated letter by writing directly into the last cluster using, for example, a low-level disk editor.

The blackmail case example of the initial implementation steps is in Listing 3.

4 Conclusion

The proposed practical approach in the cyberforensics field can also be used in a normal investigation process involving crimes not necessarily associated with information technology. Combined with an expert system (e.g. implemented in CLIPS [30]), it can also be used in training new staff in investigation techniques. The focus on hierarchical contexts as first-class values brings more understanding of the process to the investigators in cybercrime case management tools.

Future Work.

- Forensic Lucid Compiler and run-time environment.
- Prove equivalence to the FSA approach.
- Adapt/re-implement a graphical UI based on the data-flow graph tool [31] to simplify Forensic Lucid programming for not very tech-savvy investigators.

```
MrA @ es_mra
where
  evidential statement es_mra = {os_mra, os_final, os_unrelated};

  observation sequence os_mra = ($, o_unrelated_clean, $, o_blackmail, $);
  observation sequence os_final = ($, o_final);
  observation sequence os_unrelated = ($, o_unrelated, $, (Ct,0,0), $);

  observation o_final = (1, "u", "t2");
  observation o_unrelated_clean = (1, "u", "o1");

  //...

  invtrans(Q, es_mra, o_final) = backraces
  where
    // list of all possible dimensions
    observation Q = lengths box left_part box right_part;

    // events
    observation lengths = unordered {0, 1, 2};

    // symbolic labels map to human descriptions
    observation left_part = unordered {
        "u"  => "unrelated",
        "t1" => "threats-obscured part",
        "o1" => "other data (left part)"
    };

    observation right_part = unordered {
        "t2" => "threats in slack",
        "o2" => "other data (right part)"
    };

    backtraces = [ A, B, C, D, ];
    where
    ...
    end;
  end;

end;
```

Listing 3: Blackmail Case Modeling in Forensic Lucid

Acknowledgments. This research work was funded by the Faculty of Engineering and Computer Science of Concordia University, Montreal, Canada.

References

[1] Pavel Gladyshev and Ahmed Patel. Finite state machine approach to digital event reconstruction. *Digital Investigation Journal*, 2(1), 2004.

[2] Pavel Gladyshev. Finite state machine analysis of a blackmail investigation. *International Journal of Digital Evidence*, 4(1), 2005.

[3] Edward A. Ashcroft and William W. Wadge. Lucid - a formal system for writing and proving programs. *SIAM J. Comput.*, 5(3), 1976.

[4] Edward A. Ashcroft and William W. Wadge. Erratum: Lucid - a formal system for writing and proving programs. *SIAM J. Comput.*, 6((1):200), 1977.

[5] René Lalement. *Computation as Logic*. Prentice Hall, 1993. C.A.R. Hoare Series Editor. English translation from French by John Plaice.

[6] Joey Paquet and Ai Hua Wu. GIPSY – a platform for the investigation on intensional programming languages. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pages 8–14, Las Vegas, USA, June 2005. CSREA Press.

[7] Joey Paquet, Serguei A. Mokhov, and Xin Tong. Design and implementation of context calculus in the GIPSY environment. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1278–1283, Turku, Finland, July 2008. IEEE Computer Society.

[8] Joey Paquet. A multi-tier architecture for the distributed eductive execution of hybrid intensional programs. In *Proceedings of 2nd IEEE Workshop in Software Engineering of Context Aware Systems (SECASA'09)*. IEEE Computer Society, 2009. To appear.

[9] Kaiyu Wan, Vasu Alagar, and Joey Paquet. Lucx: Lucid enriched with context. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pages 48–14, Las Vegas, USA, June 2005. CSREA Press.

[10] Xin Tong. Design and implementation of context calculus in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, April 2008.

[11] The GIPSY Research and Development Group. The General Intensional Programming System (GIPSY) project. Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2002–2008. `http://newton.cs.concordia.ca/~gipsy/`, last viewed April 2008.

[12] Serguei A. Mokhov. Towards hybrid intensional programming with JLucid, Objective Lucid, and General Imperative Compiler Framework in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, October 2005. ISBN 0494102934.

[13] William Wadge and Edward Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, London, 1985.

[14] Edward Ashcroft, Anthony Faustini, Raganswamy Jagannathan, and William Wadge. *Multidimensional, Declarative Programming*. Oxford University Press, London, 1995.

[15] Edward A. Ashcroft and William W. Wadge. Lucid, a nonprocedural language with iteration. *Communication of the ACM*, 20(7):519–526, July 1977.

[16] Kaiyu Wan. *Lucx: Lucid Enriched with Context*. PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2006.

[17] Joey Paquet. *Scientific Intensional Programming*. PhD thesis, Department of Computer Science, Laval University, Sainte-Foy, Canada, 1999.

[18] Joey Paquet and Peter Kropf. The GIPSY architecture. In *Proceedings of Distributed Computing on the Web*, Quebec City, Canada, 2000.

[19] Bo Lu. *Developing the Distributed Component of a Framework for Processing Intensional Programming Languages*. PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, March 2004.

[20] Joey Paquet, Peter Grogono, and Ai Hua Wu. Towards a framework for the general intensional programming compiler in the GIPSY. In *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, Vancouver, Canada, October 2004. ACM.

[21] Emil Vassev and Joey Paquet. A generic framework for migrating demands in the GIPSY's demand-driven execution engine. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pages 29–35, Las Vegas, USA, June 2005. CSREA Press.

[22] Ai Hua Wu and Joey Paquet. Object-oriented intensional programming in the GIPSY: Preliminary investigations. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pages 43–47, Las Vegas, USA, June 2005. CSREA Press.

[23] Serguei A. Mokhov and Joey Paquet. Formally specifying and proving operational aspects of Forensic Lucid in Isabelle. Technical Report 2008-1-Ait Mohamed, Department of Electrical and Computer Engineering, Concordia University, August 2008. In Theorem Proving in Higher Order Logics (TPHOLs2008): Emerging

Trends Proceedings.

[24] Serguei A. Mokhov, Joey Paquet, and Mourad Debbabi. Formally specifying operational semantics and language constructs of Forensic Lucid. In Oliver Göbel, Sandra Frings, Detlef Günther, Jens Nedon, and Dirk Schadt, editors, *Proceedings of the IT Incident Management and IT Forensics (IMF'08)*, pages 197–216, Mannheim, Germany, September 2008. GI. LNI140.

[25] Serguei A. Mokhov. Towards syntax and semantics of hierarchical contexts in multimedia processing applications using MARFL. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1288–1294, Turku, Finland, July 2008. IEEE Computer Society.

[26] Xin Tong, Joey Paquet, and Serguei A. Mokhov. Context Calculus in the GIPSY. Unpublished, 2007.

[27] Serguei A. Mokhov, Joey Paquet, and Mourad Debbabi. Designing a language for intensional cyberforensic analysis. Unpublished, 2007.

[28] Aihua Wu, Joey Paquet, and Serguei A. Mokhov. Object-Oriented Intensional Programming: Intensional Classes Using Java and Lucid. Submitted for publication to PPPJ'09, 2009.

[29] Bram Moolenaar and Contributors. Vim the editor – Vi Improved. [online], 2007. `http://www.vim.org/`.

[30] Gary Riley. CLIPS: A tool for building expert systems. [online], 2007. `http://www.ghg.net/clips/CLIPS.html`, last viewed: December 2007.

[31] Lei Tao. Warehouse and garbage collection in the GIPSY environment. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2004.